Parallelizing CI using Docker Swarm-Mode

Akihiro Suda <suda.akihiro@lab.ntt.co.jp>
NTT Software Innovation Center
• Software Engineer at NTT Corporation

• Several talks at FLOSS community
  • FOSDEM 2016
  • ApacheCon Core North America 2016, etc.

• Docker Moby Project core maintainer

Docker project transited into Moby Project (April, 2017). Now Docker products are "downstreams" of Moby Project.
A problem in Docker/Moby project: CI is slow

https://jenkins.dockerproject.org/job/Docker-PRs/buildTimeTrend

120 min

red valley = test failed immediately 😞
How about other FLOSS projects?

(for ease of visualization, picked up some projects that use Jenkins from various categories)
Why slow CI matters?

• Blocker for reviewing/merging patches

• Discourages developers from writing tests

• Discourages developers from enabling additional testing features
  • e.g. `go test -race` (race detector) is 2-20x slower

source: https://golang.org/doc/articles/race_detector.html
Why slow CI matters?

Can result in poor implementation quality & slow development cycle 😞
Solution: Parallelize CI?

<table>
<thead>
<tr>
<th>Task</th>
<th>CPUs</th>
<th>Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>$ go test</td>
<td>$ parallel --max-procs N ...</td>
</tr>
<tr>
<td>Java (Maven)</td>
<td>$ mvn --threads</td>
<td></td>
</tr>
<tr>
<td>Python (nose)</td>
<td>$ nosetests --processes</td>
<td></td>
</tr>
</tbody>
</table>

Go
$ go test -parallel N ...

Java (Maven)
$ mvn --threads N ...

Python (nose)
$ nosetests --processes N ...

General
$ parallel --max-procs N ...
Solution: Parallelize CI?

But just doing parallelization is not enough 😞

• No isolation
  • Concurrent test tasks may race for certain shared resources
    (e.g. files under `/tmp`, TCP port, ...)

• Poor scalability
  • CPU/RAM resource limitation
  • I/O parallelism limitation
Solution: Docker (in Swarm-mode)

• Docker provides isolation
• Swarm-mode provides scalability

Parallelize & Isolate using Docker containers
✓ Isolation

Distribute across Swarm-mode
✓ Scalability
Challenge 1: Redundant setup/teardown

• For ideal isolation, each of the test functions should be encapsulated into independent containers

• But this is not optimal in actual due to setup/teardown code

```go
func TestMain(m *testing.M) {
    setUp()
    m.Run()
    tearDown()
}

func TestFoo(t *testing.T)
func TestBar(t *testing.T)
```
Optimization 1: Chunking

• Solution: execute a chunk of multiple test functions sequentially in a single container
**Challenge 2: Makespan non-uniformity**

- Test sequence is typically executed in lexicographic order (ABC...)

<table>
<thead>
<tr>
<th>Execution Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestApple1</td>
</tr>
<tr>
<td>TestApple2</td>
</tr>
<tr>
<td>TestBanana1</td>
</tr>
<tr>
<td>TestBanana2</td>
</tr>
<tr>
<td>TestCarrot1</td>
</tr>
<tr>
<td>TestCarrot2</td>
</tr>
</tbody>
</table>

- **Observation:** long-running test functions concentrate on a certain portion
  - because testing similar scenarios
Challenge 2: Makespan non-uniformity

Example: Docker itself

DockerSuite.TestBuild*

DockerSwarmSuite.Test*
Challenge 2: Makespan non-uniformity

Makespans of the chunks are likely to result in non-uniformity

Sequence

<table>
<thead>
<tr>
<th>Test1</th>
<th>Test2</th>
<th>Test3</th>
<th>Test4</th>
</tr>
</thead>
</table>

Chunks for containers

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1</td>
<td>Test3</td>
</tr>
<tr>
<td>Test2</td>
<td>Test4</td>
</tr>
</tbody>
</table>

non-uniformity

(wasted time for container 1)

(speed-up)
Optimization 2: Shuffling

Solution: shuffle the chunks

• No guarantee for optimal schedule though
Implementation

• **RPC: Funker** ([github.com/bfirsh/funker](https://github.com/bfirsh/funker))
  - FaaS-like architecture
  - Workloads are automatically balanced via Docker's built-in LB
  - No explicit task queue; when a worker is busy, the master just retries

• **Deployment:** `docker stack deploy` with Compose file
• **Easily portable to other orchestrators as well** (e.g. Kubernetes)

---

Docker Swarm-mode cluster
(typically on cloud, but even ok to use localhost as a single-node cluster)

Client

`docker stack deploy`

(on CI / laptop)

master

Funker

Built-in LB

worker.1

worker.2

worker.3
Implementation (Docker-specific part)

- Testing Docker itself requires `docker run --privileged`

- But Swarm-mode lacks support for `--privileged` at the moment
  - moby/moby#24862

- Workaround: Bind-mount `/var/run/docker.sock` into service containers, and execute `docker run --privileged` within them
Experimental result

• Evaluated my hack against the CI of Docker itself
  • Of course, this hack can be applicable to CI of other software as well

• Target: Docker 16.04-dev (git:7fb83eb7)
  • Contains 1,648 test functions

• Machine: Google Compute Engine
  n1-standard-4 instances (4 vCPUS, 15GB RAM)
Experimental result

- 20 times faster at maximum with 10 nodes
- But even effective with a single node!
## Detailed result

<table>
<thead>
<tr>
<th>Nodes</th>
<th>1 (Chunk size: 1648)</th>
<th>10 (165)</th>
<th>30 (55)</th>
<th>50 (33)</th>
<th>70 (24)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1h22m7s (=traditional)</td>
<td>15m3s</td>
<td>N/A (more than 30m)</td>
<td>5.5x (BCR 5.5x)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12m7s</td>
<td>10m12s</td>
<td>11m25s</td>
<td>13m57s</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10m16s</td>
<td>6m18s</td>
<td>5m46s</td>
<td>6m28s</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8m26s</td>
<td>4m31s</td>
<td>4m10s</td>
<td>4m20s</td>
<td></td>
</tr>
</tbody>
</table>

**Best BCR (benefit-cost ratio)**

**Fastest configuration**

Time: average of 5 runs, Graph driver: overlay2

- **BCR 5.5x**
- **BCR 4.0x**
- **BCR 2.8x**
- **BCR 2.0x**
<table>
<thead>
<tr>
<th>Nodes</th>
<th>Parallelize</th>
<th>Parallelize + Chunking</th>
<th>Parallelize + Chunking + Shuffling (= previous slide)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>14m58s</td>
<td>15m3s</td>
</tr>
<tr>
<td>2</td>
<td>more than 30m</td>
<td>10m1s</td>
<td>10m12s</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>7m32s</td>
<td>5m46s</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>6m9s</td>
<td>4m10s</td>
</tr>
</tbody>
</table>

Significantly faster

1.5x faster
## Scale-out vs Scale-up?

### Scale-out wins

- Better I/O parallelism, mutex contention, etc.

<table>
<thead>
<tr>
<th></th>
<th>Nodes</th>
<th>Total vCPUs</th>
<th>Total RAM</th>
<th>Containers</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale out</td>
<td>10</td>
<td>40</td>
<td>150GB</td>
<td>50</td>
<td>4m10s</td>
</tr>
<tr>
<td>Scale up</td>
<td>1</td>
<td>19m17s</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The code is available!

PR (merged): [docker/docker#29775](https://github.com/docker/docker/pulls/29775)

```bash
$ cd $GOPATH/src/github.com/docker/docker
$ make build-integration-cli-on-swarm
$ ./hack/integration-cli-on-swarm/integration-cli-on-swarm \
  -push-worker-image your-docker-registry/worker:latest \
  -replicas 50 \
  -shuffle
```
Is it applicable to other software as well?

Yes, generalized & simplified version available
(some Docker-specific hacks were eliminated)

[link to GitHub repository]

github.com/osrg/namazu-swarm

"Namazu Swarm"

- Namazu (鰤) means a catfish in Japanese
- Our related project: [github.com/osrg/namazu](https://github.com/osrg/namazu)
  (A tool for reproducing flaky tests and injecting faults)
- Unrelated: [www.namazu.org](http://www.namazu.org) (text search engine)
Easy to get started

Just write a Dockerfile with two labels!

```
vi Dockerfile
FROM your-project
ADD tests.txt /

LABEL \
net.osrg.namazu-swarm.v0.master.script="cat /tests.txt" \
net.osrg.namazu-swarm.v0.worker.script="sh -e -x"

? ?
Emit all the test IDs (== commands, typically) to stdout
Read a chunk of test IDs from stdin, and execute them
```
Easy Integration with CI and Clouds

Namazu Swarm itself is tested on Travis

- Travis CI
- Circle CI
- Jenkins
- (Laptop)

Planned – Kubernetes (e.g. GKE, ACS..)

Planned - ECS

Docker Swarm-mode cluster
(typically on cloud, but even ok to use localhost as a single-node cluster)
Experimental result

## Apache ZooKeeper

<table>
<thead>
<tr>
<th></th>
<th>Traditional</th>
<th>1 Node (10 containers)</th>
<th>2 Nodes (30 containers)</th>
<th>5 Nodes (50 containers)</th>
<th>10 Nodes (90 containers)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>56m7s</td>
<td>19m34s</td>
<td>17m19s</td>
<td>9m50s</td>
<td>7m52s</td>
</tr>
</tbody>
</table>

- 2.9x faster (Cost: 1x)
- 7.1x faster (Cost: 10x)

### Your own application

- Your report is welcome 😊
Future work

• Record and use the past execution history to optimize the schedule, rather than just shuffling

• Investigate deeply why scale-out wins
  • related: moby/moby#33254
Future work

• Mitigate extra overhead of pushing/pulling the image to/from the registry
  • Can take a few minutes, depending on the network condition and the amount of the cached layers on each of the nodes
  • FILEgrain: github.com/AkihiroSuda/filegrain
    • My lazy-pull extension for OCI Image Spec
    • Experimental result for `java:8` image

<table>
<thead>
<tr>
<th>Image format</th>
<th>Workload</th>
<th>Pull</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional Docker/OCI</td>
<td>Any</td>
<td>633 MB (100%)</td>
</tr>
<tr>
<td>FILEgrain</td>
<td>sh</td>
<td>4 MB (0.6%)</td>
</tr>
<tr>
<td></td>
<td>java -version</td>
<td>87 MB (14%)</td>
</tr>
<tr>
<td></td>
<td>javac Hello.java</td>
<td>136 MB (22%)</td>
</tr>
</tbody>
</table>

• P2P image distribution? e.g. IPFS

Still POC.. your contribution is welcome
Recap

- Docker Swarm-mode is effective for parallelizing CI jobs

- Introduced some techniques for optimal scheduling

- 1h22m7s (traditional testing)
  - Cost: 10x
  - Speed-up: 19.7x

- 4m10s
  - Cost: 1x
  - Speed-up: 5.5x

- 15m3s
  - 1 node
  - 4m10s
  - 10 nodes
Recap

• You can easily apply the tool to your software as well
  github.com/osrg/namazu-swarm